CS2351
Data Structures

Lecture 8:
Basic Data Structures I
About this lecture

• Once we have learnt pointers, we can now define some basic, but very useful, data structures

• We will introduce three of them here:
  1. List
  2. Queue (also called FIFO queue)
  3. Stack (also called LIFO queue)
List
List

- A list (or linked list) is a data structure to represent a sequence of items, one after the other

A list of people
List

• Each item in the list points at the item immediately after it
• Usually, we keep an extra pointer, called head, to point at the first item
List

• Once the head of a list is known, we can traverse the list (from the beginning to the end) in linear time

• Usually, an item in the list is called a node
Implementing a List in C

• In C, we can first define a new type to represent a node:

```c
struct node {
    ...
    ...
} ;
```
Implementing a List in C

• Since each node points to the next one, so we should have:

```c
struct node {
    ...
    struct node *next;
};
```
Implementing a List in C

• Also, each node may contain some info
• Ex: To represent a list of people, a node may need to store the name of a person
• In this case, the definition may look like:

```c
struct node {
    char name[80];
    struct node *next;
};
```
Implementing a List in C

- Once the definition of a node is done, we can create a list

```c
struct node x, y, *head;

strcpy(x.name, "John");
strcpy(y.name, "Mary");
head = &x;  x.next = &y;
y.next = 0;
```

```
head  John  Mary
```
Implementing a List in C

- Also, we can traverse a list easily

```c
struct node *current;
current = head;
while (current != 0)
{
    printf("%s\n", (*current).name);
    current = (*current).next;
}
```

![Diagram of a list with nodes John and Mary connected by arrows to represent traversal.](Diagram.png)
Remark 1

- Recall that we have written something like
  \[
  y\text{.next} = 0 ;
  \]
  to specify that \( y \) points to nothing
- In \textit{C}, we often use \texttt{NULL} to replace 0, so as to show it indeed represents a location
- Then, we will write something like:
  \[
  y\text{.next} = \texttt{NULL} ;
  \]
  \[
  \text{while} \ ( \texttt{current} != \texttt{NULL} ) \ { \ldots } \]
Remark 2

• Recall that we have written something like

\[
\text{current} = (\ast\text{current}).\text{next} ;
\]

• The right hand side looks clumsy

• In \textit{C}, we have a shorthand notation \( \rightarrow \) (which looks like an arrow) to simply

• Instead of \( (\ast\text{current}).\text{next} \), we write

\[
\text{current} = \text{current}\rightarrow\text{next} ;
\]

• In general, \( (\ast\text{ptr}).\text{val} \) is exactly \( \text{ptr}\rightarrow\text{val} \)
Insert in a List

- Suppose we have a **pointer** that points at a node $X$ in the list
- Then, we can easily use this **pointer** to insert an extra node after $X$ (How?)
Insert in a List

• Let \texttt{current} be the \texttt{pointer} that specifies where to insert
• Let \texttt{y} be the extra node to be inserted
• Then, we can perform insert as follows:

\begin{verbatim}
y.next = current->next ;
current->next = &y ;
\end{verbatim}

• Thus, if we know where to insert, only \textit{O}(1) time is required!
Delete in a List

• Similarly, if there is a pointer that points at a node \( X \), we can delete a node after \( X \)

```c
if ( current->next != NULL )
{
    current->next = current->next->next ;
}
```

• Thus, if we know where to delete, only \( O(1) \) time is required!
Remarks for List Updates

Q: If we have a pointer that points at $X$, can we insert a node before $X$?

A: Yes. We traverse from head, until we find a node $Y$ that points to $X$ in the list
- $Y$ must be the node before $X$
- After that, we insert an extra node after $Y$

Q: Then, can we delete a node before $X$?

A: Yes. (How?)
Remarks for List Updates

• Insert/delete before a node is tedious
  - In the worst case, it takes linear time!
• If we want to support such operations, we may use doubly linked list, so that each node has two pointers
  - one to previous node, one to next node

```c
struct node {
    ...
    struct node *prev, *next;
};
```
Queue
A queue is a special kind of list where insertion is always at the end, and deletion is always at the front.

Deletion always at the front
Insertion always at the end
Deletion in a Queue

• Since we have the head of a list, we can perform deletion easily (in $O(1)$ time)

```c
if ( head != NULL )
{
    head = head->next ;
}
```

• Here, we assume that in an empty queue, head is set to NULL
Insertion in a Queue

• To speed up the insertion, we will keep an extra pointer, called `tail`, that points at the last item in a queue

• Then, we can insert a node `y` in \( O(1) \) time without traversing the whole queue:

```c
if ( head != NULL )
{
    tail->next = &y ;
    tail = &y;
}
```
Remarks for Queues

• Because we now maintain both head and tail pointers, we need to be careful in the boundary cases (when we insert a node in an empty queue, or delete the node to make the queue empty)

• The insert/delete operations in a queue are often called enqueue/dequeue

• Queue is also known as FIFO (first in first out) queue
Remarks for Queues

• To summarize the above, we may write a function for `enqueue` as follows:

```c
void enqueue( struct node **head, 
              struct node **tail, struct node *y )
{
    if ( (*head) != NULL ) // if not empty
    { (*tail)->next = y ; (*tail) = y; } 
    else
    { (*head) = (*tail) = y ; } 
}
```
Stack
• A stack is a special kind of list where insertion/deletion are always at the end
• Such an end is often called top

Stack

Insertion/Deletion always at the top
Deletion in a Stack

• We maintain a pointer, called top, to points at the top of the stack
• Since after deletion, we need to update top, each node should point at the previous node in the stack
• Then deletion is easily done (in O(1) time):

```c
if ( top != NULL ) // if not empty
{
    top = top->prev ;
}
```
Insertion in a Stack

• Insertion of a node $y$ into the stack is also easy (done in $O(1)$ time)

$$y.\text{prev} = \text{top} ;$$
$$\text{top} = &y ;$$

Remarks:

• Insertion/Deletion operations in a stack are often called Push/Pop

• Stack is also known as LIFO (Last in first out) queue
Practical Implementation

• In practice, we normally use an array to represent Queue or Stack

  Advantage: Each operation is faster
  (no need to keep next/prev pointers)

  Disadvantage: Wasted space / Overflow

• We will discuss further in the tutorial